PANDAS FOR BEGINNERS

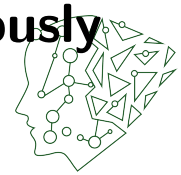
AIOTLAB, September 2023

Table of Contents

Table of Contents

Getting Started

- **pandas** contains **data structures** and **data manipulation tools** designed to make *data cleaning* and *analysis* fast and convenient in Python
- **pandas** is often used in tandem with:
  - Numerical computing tools (**NumPy** and **SciPy**)
  - Analytical libraries like (**statsmodels** and **scikit-learn**)
  - Data visualization libraries (**matplotlib**)
- **pandas** adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing *without* **for** loops
- While **pandas** adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with **tabular** or **heterogeneous data**
  - **NumPy**, by contrast, is *best* suited for working with **homogeneously** typed **numerical** array data



Getting Started

- The following *import conventions* for **NumPy** and **pandas** are commonly used:

```
1  import numpy as np
2
3  import pandas as pd
```

- It is easier to import Series and DataFrame into the local namespace since they are so frequently used

```
1  from pandas import Series, DataFrame
```

Table of Contents

Series

- A **Series** is a **one-dimensional** array-like object containing a sequence of *values* (of similar types to NumPy types) of the **same type** and an associated array of *data labels*, called its **index**

```
obj = pd.Series([4, 7, -5, 3])

obj

Output:
1    7
2   -5
3    3
dtype: int64
```

Series

- The string representation of a Series displayed interactively shows the index on the left and the values on the right
  - Since we did not specify an index for the data, a default one consisting of the integers **0** through **N - 1** (where **N** is the length of the data) is created
  - Can get the array representation and index object of the Series via its **array** and **index** attributes, respectively
  - The result of the **.array** attribute is a **PandasArray** which usually wraps a NumPy array but can also contain special extension array types

```
obj.array
Output:
<NumpyExtensionArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

obj.index
Output: RangeIndex(start=0, stop=4, step=1)
```

Series

- Often, we'll want to create a Series with an **index** identifying each data point with a **label**

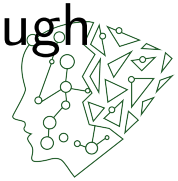
```
obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

obj2
Output:
d    4
b    7
a   -5
c    3
dtype: int64

obj2.index
Output: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Series

- Compared with NumPy arrays, we can use labels in the index when selecting *single* values or a *set* of values

```
obj2["a"]
Output: -5

obj2["d"] = 6

obj2[["c", "a", "d"]]
Output:
c    3
a   -5
d    6
dtype: int64
```

- Here **["c", "a", "d"]** is interpreted as a *list* of *indices*, even though it contains strings instead of integers



Series

- Using NumPy functions or NumPy-like operations, such as filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve the **index-value link**

```
obj2[obj2 > 0]
Output:
d    6
b    7
c    3
dtype: int64

obj2 * 2
Output:
d    12
b    14
a   -10
c     6
dtype: int64
```

```
np.exp(obj2)
Output:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Series

- Can think of Series as a *fixed-length*, *ordered* **dictionary**, as it is a mapping of index values to data values => Can be used in many contexts where we might use a dictionary

```
"b" in obj2
Output: True

"e" in obj2
Output: False
```

Series

- Should we have data contained in a Python dictionary, we can create a Series from it by passing the dictionary
- A Series can be converted back to a dictionary with its **to_dict** method

```
sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

obj3 = pd.Series(sdata)

obj3
Output:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64

obj3.to_dict()
Output: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

Series

- When only passing a dictionary, the index in the resulting Series will respect the order of the keys according to the dictionary's **keys** method, which depends on the key insertion order
  - Can override this by passing an **index** with the dictionary keys in the order we want them to appear in the resulting Series

Series

- Here, three values found in **sdata** were placed in the appropriate locations
  - But since no value for **"California"** was found, it appears as **NaN** (Not a Number), which is considered in pandas to mark missing or *NA* values
  - Since **"Utah"** was not included in **states**, it is excluded from the resulting object

```
sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}
states = ["California", "Ohio", "Oregon", "Texas"]

obj4 = pd.Series(sdata, index=states)

obj4
Output:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Series

- The **isna** and **notna** functions in pandas should be used to detect missing data
  - Series also has these as **instance methods**

```
pd.isna(obj4)
Output:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

pd.notna(obj4)
Output:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

```
obj4.isna()
Output:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

obj4.notna()
Output:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series

- A useful Series feature for many applications is that it **automatically aligns** by **index label** in arithmetic operations
  - This feature is similar to a *join operation* in databases

```
obj3
Output:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64

obj4
Output:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
obj3 + obj4
Output:
California        NaN
Ohio          70000.0
Oregon        32000.0
Texas        142000.0
Utah              NaN
dtype: float64
```

Series

- Both the Series **object** itself and its **index** have a **name** attribute
  - It integrates with other areas of pandas functionality

```
obj4.name = "population"

obj4.index.name = "state"

obj4
Output:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series

- A Series's index can be altered in place by assignment

```
obj
Output:
0    4
1    7
2   -5
3    3
dtype: int64

obj.index = ["Bob", "Steve", "Jeff", "Ryan"]

obj
Output:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

- A **DataFrame** represents a **rectangular table** of data and contains an **ordered**, **named** collection of columns, each of which can be a **different value type** (numeric, string, Boolean, etc.)
  - The DataFrame has both a row and column index
  - Can be thought of as a dictionary of Series all sharing the same index
- **Note**: While DataFrame is physically two-dimensional, we can use it to represent *higher dimensional* data in a tabular format using:
  - Hierarchical indexing,
  - and an ingredient in some of the more advanced data-handling features in pandas

DataFrame

- There are many ways to **construct** a DataFrame, though one of the **most common** is from a **dictionary** of equal-length lists or NumPy arrays
  - The resulting DataFrame will have its index assigned automatically, as with Series
  - The columns are placed according to the order of the keys in **data** (depends on their insertion order in the dictionary)

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)

frame
Output:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

DataFrame

- For large DataFrames:
  - The **head** method selects only the **first 5 rows**
  - The **tail** method selects only the **last 5 rows**

```
frame.head()
Output:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

```
frame.tail()
Output:
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

DataFrame

- If we specify a sequence of columns, the DataFrame's columns will be arranged in that order

```
pd.DataFrame(data, columns=["year", "state", "pop"])
Output:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

DataFrame

- If we pass a *column* that *isn't contained* in the dictionary, it will appear with **missing values** in the result

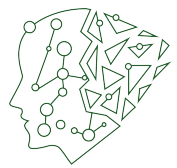
```
frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])

frame2
Output:
   year   state  pop debt
0  2000    Ohio  1.5  NaN
1  2001    Ohio  1.7  NaN
2  2002    Ohio  3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
5  2003  Nevada  3.2  NaN

frame2.columns
Output: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame

- A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by using the dot attribute notation
  - **Note 1**: Attribute-like access (e.g., **frame2.year**) and tab completion of column names in IPython are provided as a convenience
  - **Note 2**: **frame2[column]** works for any column name, but **frame2.column** works only when the column name is a valid Python variable name and does not conflict with any of the method names in DataFrame
    - E.g., if a column's name contains whitespace or symbols other than underscores, it cannot be accessed with the dot attribute method
  - The returned Series have the **same index** as the DataFrame, and their **name** attribute has been **appropriately set**

```
frame2["state"]
Output:
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
```

```
frame2.year
Output:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```



DataFrame

- Rows can also be retrieved by position or name with the special **iloc** and **loc** attributes

```
frame2.loc[1]
Output:
year     2001
state    Ohio
pop       1.7
debt      NaN
Name: 1, dtype: object

frame2.iloc[2]
Output:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: 2, dtype: object
```

DataFrame

- Columns can be modified by assignment (e.g., the empty **debt** column could be assigned a scalar value or an array of values)

```
frame2["debt"] = 16.5

frame2
Output:
   year   state  pop  debt
0  2000    Ohio  1.5  16.5
1  2001    Ohio  1.7  16.5
2  2002    Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5
```

```
frame2["debt"] = np.arange(6.)

frame2
Output:
   year   state  pop  debt
0  2000    Ohio  1.5   0.0
1  2001    Ohio  1.7   1.0
2  2002    Ohio  3.6   2.0
3  2001  Nevada  2.4   3.0
4  2002  Nevada  2.9   4.0
5  2003  Nevada  3.2   5.0
```

DataFrame

- When we are assigning lists or arrays to a column, the value's length must match the length of the DataFrame
  - If we assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any index values not present
  - **Note**: Assigning a column that *doesn't exist* will create a *new* column

```
val = pd.Series([-1.2, -1.5, -1.7], index=[2, 4, 5])

frame2["debt"] = val

frame2
Output:
   year   state  pop  debt
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2002    Ohio  3.6  -1.2
3  2001  Nevada  2.4   NaN
4  2002  Nevada  2.9  -1.5
5  2003  Nevada  3.2  -1.7
```

DataFrame

- The **del** keyword will delete columns like with a dictionary
  - Caution 1: New columns cannot be created with the **frame2.eastern** dot attribute notation
  - Caution 2: The column returned from indexing a DataFrame is a *view* on the underlying data, not a copy
    - Thus, any in-place modifications to the Series will be reflected in the DataFrame
    - The column can be explicitly copied with the Series's **copy** method

```
frame2["eastern"] = frame2["state"] == "Ohio"

frame2
Output:
   year   state  pop  debt  eastern
0  2000    Ohio  1.5   NaN     True
1  2001    Ohio  1.7   NaN     True
2  2002    Ohio  3.6  -1.2     True
3  2001  Nevada  2.4   NaN    False
4  2002  Nevada  2.9  -1.5    False
5  2003  Nevada  3.2  -1.7    False

del frame2["eastern"]

frame2.columns
Output: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame

- Another common form of data is a nested dictionary of dictionaries
  - If the nested dictionary is passed to the DataFrame, pandas will interpret the outer dictionary keys as the columns, and the inner keys as the row indices
  - Can **transpose** the DataFrame (swap rows and columns) with similar syntax to a NumPy array
  - **Warning**: Transposing discards the column data types if the columns do not **all** have the **same data type**, so transposing and then transposing back may lose the previous type information
  - The keys in the inner dictionaries are combined to form the index in the result
    - This isn't true if an explicit index is specified

DataFrame

```
populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
               "Nevada": {2001: 2.4, 2002: 2.9}}

frame3 = pd.DataFrame(populations)

frame3
Output:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2002   3.6     2.9

frame3.T
Output:
         2000  2001  2002
Ohio      1.5   1.7   3.6
Nevada    NaN   2.4   2.9

pd.DataFrame(populations, index=[2001, 2002, 2003])
Output:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
2003   NaN     NaN
```

DataFrame

- Dictionaries of Series

```
pdata = {"Ohio": frame3["Ohio"][:-1],
         "Nevada": frame3["Nevada"][:2]}

pd.DataFrame(pdata)
Output:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
```

- If a DataFrame's **index** and **columns** have their **name** attributes set, these will also be displayed. Unlike Series, DataFrame does NOT have a **name** attribute

```
frame3.index.name = "year"
frame3.columns.name = "state"

frame3
Output:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

DataFrame

- DataFrame's **to_numpy** method returns the data contained in the DataFrame as a two-dimensional ndarray

```
frame3.to_numpy()

Output: array([[1.5, nan],
               [1.7, 2.4],
               [3.6, 2.9]])
```

- If the DataFrame's columns are different data types, the data type of the returned array will be chosen to accommodate all of the columns

```
frame2.to_numpy()

Output: array([[2000, 'Ohio', 1.5, nan],
               [2001, 'Ohio', 1.7, nan],
               [2002, 'Ohio', 3.6, -1.2],
               [2001, 'Nevada', 2.4, nan],
               [2002, 'Nevada', 2.9, -1.5],
               [2003, 'Nevada', 3.2, -1.7]], dtype=object)
```

DataFrame

- **Possible data inputs** to the DataFrame constructor:

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
Dictionary of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dictionary of arrays" case
Dictionary of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
Dictionary of dictionaries	Each inner dictionary becomes a column; keys are unioned to form the row index as in the "dictionary of Series" case
List of dictionaries or Series	Each item becomes a row in the DataFrame; unions of dictionary keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values are missing in the DataFrame result

Index Objects

- pandas's **Index objects** are responsible for holding:
  - The **axis labels** (including a DataFrame's column names)
  - Other **metadata** (like the axis name or names)
- Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index

```
obj = pd.Series(np.arange(3), index=["a", "b", "c"])

index = obj.index

index
Output: Index(['a', 'b', 'c'], dtype='object')

index[1:]
Output: Index(['b', 'c'], dtype='object')
```

Index Objects

- Index objects are **immutable** => Can't be modified by the user

```
index[1] = "d"

Output: TypeError: Index does not support mutable operations
```

- Immutability makes it *safer* to share Index objects among data structures
- **Caution**: Some users will not often take advantage of the capabilities provided by an Index, but because some operations will yield results containing indexed data, it's important to understand how they work

Index Objects

```
labels = pd.Index(np.arange(3))

labels
Output: Index([0, 1, 2], dtype='int32')

obj2 = pd.Series([1.5, -2.5, 0], index=labels)

obj2
Output:
0    1.5
1   -2.5
2    0.0
dtype: float64

obj2.index is labels
Output: True
```

Index Objects

- In addition to being array-like, an Index also behaves like a fixed-size set

```
frame3
Output:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9

frame3.columns
Output: Index(['Ohio', 'Nevada'], dtype='object', name='state')

"Ohio" in frame3.columns
Output: True

2003 in frame3.index
Output: False
```

- Unlike Python sets, a pandas Index can contain **duplicate** *labels*
  - Selections with duplicate labels will select all occurrences of that label

```
pd.Index(["foo", "foo", "bar", "bar"])

Output: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Index Objects

- Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains
- Some Index methods and properties

Method/Property	Description
`append()`	Concatenate with additional Index objects, producing a new Index
`difference()`	Compute set difference as an Index
`intersection()`	Compute set intersection
`union()`	Compute set union
`isin()`	Compute Boolean array indicating whether each value is contained in the passed collection
`delete()`	Compute new Index with element at Index `i` deleted
`drop()`	Compute new Index by deleting passed values
`insert()`	Compute new Index by inserting element at Index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique()`	Compute the array of unique values in the Index

Table of Contents

Reindexing

- An important method on pandas objects is **reindex**
  - Create a new object with the values *rearranged* to align with the new index
  - Rearranges the data according to the new index, introducing *missing values* if any index values were not already present

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])

obj
Output: d    4.5
        b    7.2
        a   -5.3
        c    3.6
        dtype: float64

obj2 = obj.reindex(["a", "b", "c", "d", "e"])

obj2
Output: a   -5.3
        b    7.2
        c    3.6
        d    4.5
        e    NaN
        dtype: float64
```

Reindexing

- For ordered data like time **series**, you may want to do some interpolation or filling of values when reindexing
  - The **method** option allows us to do this, using a method such as **ffill**, which **forward-fills** the values

```
obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

obj3
Output: 0      blue
        2    purple
        4    yellow
        dtype: object

obj3.reindex(np.arange(6), method="ffill")
Output: 0      blue
        1      blue
        2    purple
        3    purple
        4    yellow
        5    yellow
        dtype: object
```

Reindexing

- With **DataFrame**, **reindex** can alter the (row) index, columns, or both
  - When passed only a sequence, it reindexes the rows in the result

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=["a", "c", "d"],
                     columns=["Ohio", "Texas", "California"])

frame
Output:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

frame2 = frame.reindex(index=["a", "b", "c", "d"])

frame2
Output:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

Reindexing

- The columns can be reindexed with the **columns** keyword
  - Because **"Ohio"** was not in **states**, the data for that column is dropped from the result

```
states = ["Texas", "Utah", "California"]

frame.reindex(columns=states)
Output:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

- Reindex an axis by passing the new axis labels as a positional argument and specify the axis to reindex with the **axis** keyword

```
frame.reindex(states, axis="columns")

Output:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Reindexing

- Can also reindex by using the **loc** operator, and many users prefer to always do it this way
  - This works only if all of the new index labels *already exist* in the DataFrame (whereas **reindex** will insert *missing data* for new labels)

```
frame.loc[["a", "d", "c"], ["California", "Texas"]]

Output:
   California  Texas
a           2      1
d           8      7
c           5      4
```

Reindexing

- **reindex** function **arguments**:

Argument	Description
`labels`	New sequence to use as an index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`index`	Use the passed sequence as the new index labels.
`columns`	Use the passed sequence as the new column labels.
`axis`	The axis to reindex, whether `"index"` (rows) or `"columns"`. The default is `"index"`. You can alternately do `reindex(index=new_labels)` or `reindex(columns=new_labels)`.
`method`	Interpolation (fill) method; `"ffill"` fills forward, while `"bfill"` fills backward.

Reindexing

- **reindex** function **arguments** (cont.):

`fill_value`	Substitute value to use when introducing missing data by reindexing. Use `fill_value="missing"` (the default behavior) when you want absent labels to have null values in the result.
`limit`	When forward filling or backfilling, the maximum size gap (in number of elements) to fill.
`tolerance`	When forward filling or backfilling, the maximum size gap (in absolute numeric distance) to fill for inexact matches.
`level`	Match simple Index on level of MultiIndex; otherwise select subset of.
`copy`	If `True`, always copy underlying data even if the new index is equivalent to the old index; if `False`, do not copy the data when the indexes are equivalent.

Dropping Entries from an Axis

- If we already have an index array or list without entries, we can use the **reindex** method or **.loc**-based indexing
- The **drop** method will return a new object with the indicated value or values deleted from an axis

```
obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])
obj
Output: a    0.0
        b    1.0
        c    2.0
        d    3.0
        e    4.0
        dtype: float64
```

```
new_obj = obj.drop("c")
new_obj
Output: a    0.0
        b    1.0
        d    3.0
        e    4.0
        dtype: float64
```

```
obj.drop(["d", "c"])
Output: a    0.0
        b    1.0
        e    4.0
        dtype: float64
```

Dropping Entries from an Axis

- With **DataFrame**, index values can be deleted from either axis
  - Calling **drop** with a sequence of labels will drop values from the row labels (axis 0)
  - To drop labels from the columns, instead use the **columns** keyword

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=["Ohio", "Colorado", "Utah", "New York"],
                    columns=["one", "two", "three", "four"])
data
Output:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
data.drop(index=["Colorado", "Ohio"])
Output:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

```
data.drop(columns=["two"])
Output:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

Dropping Entries from an Axis

- Can also drop values from the columns by passing **axis=1** (which is like NumPy) or **axis="columns"**

```
data.drop("two", axis=1)
Output:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

data.drop(["two", "four"], axis="columns")
Output:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Indexing, Selection, and Filtering

- Series indexing (**obj[...]**) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers

```
obj = pd.Series(np.arange(4.),
                index=["a", "b", "c", "d"])
obj
Output: a    0.0
        b    1.0
        c    2.0
        d    3.0
        dtype: float64

obj["b"]
Output: 1.0

obj[1]
Output: 1.0

obj[2:4]
Output: c    2.0
        d    3.0
        dtype: float64
```

```
obj[["b", "a", "d"]]
Output: b    1.0
        a    0.0
        d    3.0
        dtype: float64

obj[[1, 3]]
Output: b    1.0
        d    3.0
        dtype: float64

obj[obj < 2]
Output: a    0.0
        b    1.0
        dtype: float64
```

Indexing, Selection, and Filtering

- While we can select data by label this way, the preferred way to select index values is with the special **loc** operator

```
obj.loc[["b", "a", "d"]]

Output: b    1.0
        a    0.0
        d    3.0
        dtype: float64
```

- The reason to prefer **loc** is because of the different treatment of integers when indexing with **[ ]**
  - Regular **[ ]**-based indexing will treat integers as labels if the index contains integers, so the behavior differs depending on the data type of the index

Indexing, Selection, and Filtering

```
obj1 = pd.Series([1, 2, 3],
                 index=[2, 0, 1])
obj2 = pd.Series([1, 2, 3],
                 index=["a", "b", "c"])

obj1
Output: 2    1
        0    2
        1    3
        dtype: int64

obj2
Output: a    1
        b    2
        c    3
        dtype: int64
```

```
obj1[[0, 1, 2]]
Output: 0    2
        1    3
        2    1
        dtype: int64

obj2[[0, 1, 2]]
Output: a    1
        b    2
        c    3
        dtype: int64
```

Indexing, Selection, and Filtering

- When using **loc**, the expression **obj.loc[[0, 1, 2]]** will **fail** when the index does NOT contain integers

```
obj2.loc[[0, 1]]

Output:
KeyError                                  Traceback (most recent call last)
... \pandas.ipynb Cell 98 line 1
----> 1 obj2.loc[[0, 1]]

^ LONG EXCEPTION ABBREVIATED ^

KeyError: 'None of [Index([0, 1], dtype='int32')] are in the [index]'
```

Indexing, Selection, and Filtering

- Since **loc** operator indexes exclusively with **labels**, there is also an **iloc** operator that indexes exclusively with **integers** to work consistently whether or not the index contains integers

```
obj1.iloc[[0, 1, 2]]
Output: 2    1
        0    2
        1    3
        dtype: int64

obj2.iloc[[0, 1, 2]]
Output: a    1
        b    2
        c    3
        dtype: int64
```

Indexing, Selection, and Filtering

- **Caution**: You can also *slice* with *labels*, but it works differently from normal Python slicing in that the **endpoint** is **inclusive**
- Assigning values using these methods (**loc** and **iloc**) modifies the corresponding section of the Series
- **Common newbie error**: trying to call **loc** or **iloc** like *functions* rather than "indexing into" them with square brackets
  - The square bracket **[ ]** notation is used to enable slice operations and to allow for indexing on multiple axes with DataFrame objects

```
obj2.loc["b":"c"]
Output: b    2
        c    3
        dtype: int64

obj2.loc["b":"c"] = 5

obj2
Output: a    1
        b    5
        c    5
        dtype: int64
```

Indexing, Selection, and Filtering

- Indexing into a **DataFrame** retrieves one or more columns either with a single value or sequence

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=["Ohio", "Colorado", "Utah", "New York"],
                    columns=["one", "two", "three", "four"])

data
Output:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
data["two"]
Output: Ohio         1
        Colorado     5
        Utah         9
        New York    13
        Name: two, dtype: int32
```

```
data[["three", "one"]]
Output:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing, Selection, and Filtering

- Indexing like this has a few special cases:
  - *Case 1*: Slicing or selecting data with a Boolean array
    - The row selection syntax **data[:2]** is provided as a convenience
    - Passing a single element or a list to the **[ ]** operator selects columns

```
data[:2]
Output:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
data[data["three"] > 5]
Output:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

  - *Case 2*: Indexing with a Boolean DataFrame, such as one produced by a scalar comparison

```
data < 5
Output:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False
```

```
data[data < 5] = 0

data
Output:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Indexing, Selection, and Filtering

- Like Series, **DataFrame** has special attributes **loc** and **iloc** for *label*-based and *integer*-based indexing, respectively
  - Since DataFrame is **two-dimensional**, we can select a subset of the rows and columns with NumPy-like notation using either axis labels (**loc**) or integers (**iloc**)

```
data
Output:
            one  two  three  four
Ohio          0    0      0     0
Colorado      0    5      6     7
Utah          8    9     10    11
New York     12   13     14    15
```

```
data.loc["Colorado"]
Output: one      0
        two      5
        three    6
        four     7
        Name: Colorado, dtype: int32
```

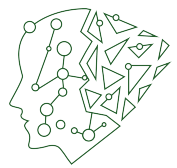
Indexing, Selection, and Filtering

- The result of selecting a single row is a Series with an index that contains the DataFrame's column labels
  - To select multiple roles, creating a new DataFrame, pass a sequence of labels

```
data.loc[["Colorado", "New York"]]
Output:
           one  two  three  four
Colorado     0    5      6     7
New York    12   13     14    15
```

  - Can combine both row and column selection in **loc** by separating the selections with a **comma**

```
data.loc["Colorado", ["two", "three"]]
Output: two      5
        three    6
        Name: Colorado, dtype: int32
```



Indexing, Selection, and Filtering

- Can perform some similar selections with integers using **iloc**

```
data.iloc[2]
Output: one       8
        two       9
        three    10
        four     11
        Name: Utah, dtype: int32

data.iloc[[2, 1]]
Output:
            one  two  three  four
Utah          8    9     10    11
Colorado      0    5      6     7
```

```
data.iloc[2, [3, 0, 1]]
Output: four    11
        one      8
        two      9
        Name: Utah, dtype: int32

data.iloc[[1, 2], [3, 0, 1]]
Output:
             four  one  two
Colorado        7    0    5
Utah           11    8    9
```

Indexing, Selection, and Filtering

- Both indexing functions (**loc** and **iloc**) work with slices in addition to single labels or lists of labels

```
data.loc[:"Utah", "two"]

Output: Ohio        0
        Colorado    5
        Utah        9
        Name: two, dtype: int32
```

```
data.iloc[:, :3][data.three > 5]

Output:
            one  two  three
Colorado      0    5      6
Utah          8    9     10
New York     12   13     14
```

- **Caution**: Boolean arrays can be used with **loc** but NOT **iloc**

```
data.loc[data.three ⩾ 2]

Output:
            one  two  three  four
Colorado      0    5      6     7
Utah          8    9     10    11
New York     12   13     14    15
```

Indexing, Selection, and Filtering

- Indexing options with DataFrame:

Type	Notes
`df[column]`	Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion)
`df.loc[rows]`	Select single row or subset of rows from the DataFrame by label
`df.loc[:, cols]`	Select single column or subset of columns by label
`df.loc[rows, cols]`	Select both row(s) and column(s) by label
`df.iloc[rows]`	Select single row or subset of rows from the DataFrame by integer position
`df.iloc[:, cols]`	Select single column or subset of columns by integer position
`df.iloc[rows, cols]`	Select both row(s) and column(s) by integer position
`df.at[row, col]`	Select a single scalar value by row and column label
`df.iat[row, col]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels

Indexing, Selection, and Filtering

- **Caution 1**: To avoid ambiguity and certain **pitfalls** regarding *Integer Indexing*, it is best to always prefer indexing with **loc** and **iloc**
- **Caution 2**: A good rule of thumb is to avoid *Chain Indexing* when doing **assignments**
  - There are other cases where pandas will generate **SettingWithCopyWarning** that have to do with chained indexing

```
data.loc[data.three == 5]["three"] = 6

Output:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

```
data.loc[data.three == 5, "three"] = 6

data
Output:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      6     5
New York    3    3      3     3
```

Arithmetic and Data Alignment

- pandas can make it much simpler to work with objects that have *different indexes*
  - E.g., when we add objects, if any index pairs are *not the same*, the respective index in the result will be the **union** of the *index pairs*
- The internal data alignment introduces **missing values** in the label locations that *don't overlap*
  - Missing values will then propagate in further arithmetic computations

Arithmetic and Data Alignment

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5],
               index=["a", "c", "d", "e"])

s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=["a", "c", "e", "f", "g"])

s1
Output: a    7.3
        c   -2.5
        d    3.4
        e    1.5
        dtype: float64

s2
Output: a   -2.1
        c    3.6
        e   -1.5
        f    4.0
        g    3.1
        dtype: float64
```

```
s1 + s2
Output: a    5.2
        c    1.1
        d    NaN
        e    0.0
        f    NaN
        g    NaN
        dtype: float64
```

Arithmetic and Data Alignment

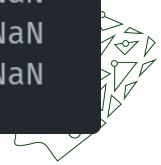
- In the case of **DataFrame**, alignment is performed on **both** rows and columns
  - Since the **"c"** and **"e"** columns are not found in both DataFrame objects, they appear as missing in the result
  - The same holds for the rows with labels that are not common to both objects

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
                   index=["Ohio", "Texas", "Colorado"])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
                   index=["Utah", "Ohio", "Texas", "Oregon"])
```

```
df1
Output:
             b    c    d
Ohio       0.0  1.0  2.0
Texas      3.0  4.0  5.0
Colorado   6.0  7.0  8.0
```

```
df2
Output:
             b    d     e
Utah       0.0  1.0   2.0
Ohio       3.0  4.0   5.0
Texas      6.0  7.0   8.0
Oregon     9.0  10.0  11.0
```

```
df1 + df2
Output:
             b   c    d   e
Colorado   NaN NaN  NaN NaN
Ohio       3.0 NaN  6.0 NaN
Oregon     NaN NaN  NaN NaN
Texas      9.0 NaN 12.0 NaN
Utah       NaN NaN  NaN NaN
```



Arithmetic and Data Alignment

- If we add DataFrame objects with no column or row labels in common, the result will contain all nulls

```
df1 = pd.DataFrame({"A": [1, 2]})
df2 = pd.DataFrame({"B": [3, 4]})

df1
Output:
   A
0  1
1  2

df2
Output:
   B
0  3
1  4
```

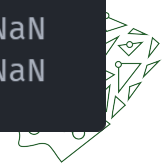
```
df1 + df2
Output:
     A   B
0 NaN NaN
1 NaN NaN
```

Arithmetic and Data Alignment

- In arithmetic operations between differently indexed objects, we might want to fill with a special value, like 0, when an axis label is found in one object but not the other

```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
                   columns=list("abcd"))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
                   columns=list("abcde"))

df1
Output:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

df2
Output:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   6.0   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

```
df1 + df2
Output:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0  11.0  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```



Arithmetic and Data Alignment

- Using the **add** method on **df1**, and pass **df2** and an argument to **fill_value**, which substitutes the passed value for any missing values in the operation

```
df1.add(df2, fill_value=0)

Output:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0  11.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

Arithmetic and Data Alignment

- Flexible arithmetic methods for Series and DataFrame:
  - Each has a counterpart, starting with the letter **r**, that has arguments reversed

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Arithmetic and Data Alignment

```
1 / df1

Output:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

```
df1.rdiv(1)

Output:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

- Relatedly, when reindexing a Series or DataFrame, we can also specify a different fill value

```
df1.reindex(columns=df2.columns, fill_value=0)

Output:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

Arithmetic and Data Alignment

- As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined
- Consider the difference between a two-dimensional array and one of its rows
  - When we subtract **arr[0]** from **arr**, the subtraction is performed once for each row (referred to as **Broadcasting**)

```
arr = np.arange(12.).reshape((3, 4))

arr
Output: array([[ 0.,  1.,  2.,  3.],
               [ 4.,  5.,  6.,  7.],
               [ 8.,  9., 10., 11.]])

arr[0]
Output: array([0., 1., 2., 3.])

arr - arr[0]
Output: array([[0., 0., 0., 0.],
               [4., 4., 4., 4.],
               [8., 8., 8., 8.]])
```

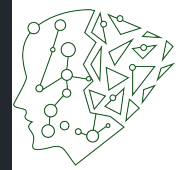
Arithmetic and Data Alignment

- Operations between a DataFrame and a Series are similar
  - By default, arithmetic between DataFrame and Series matches the index of the Series on the columns of the DataFrame, **broadcasting** down the rows

```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
                     index=["Utah", "Ohio", "Texas", "Oregon"])
series = frame.iloc[0]
```

```
frame
Output:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

series
Output: b    0.0
        d    1.0
        e    2.0
        Name: Utah, dtype: float64
```

```
frame - series
Output:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```



Arithmetic and Data Alignment

- If an **index** value is **not found** in either the DataFrame's columns or the Series's index, the objects will be **reindexed** to form the **union**

```
series2 = pd.Series(np.arange(3), index=["b", "e", "f"])

series2
Output: b    0
        e    1
        f    2
        dtype: int32

frame + series2
Output:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

Arithmetic and Data Alignment

- If we want to instead **broadcast** over the *columns*, **matching** on the *rows*, we have to use one of the **arithmetic methods** and specify to match over the index
  - The axis that we pass is the **axis to match on**
  - In this case we mean to match on the DataFrame's row index (**axis="index"**) and broadcast across the columns

```
series3 = frame["d"]

frame
Output:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

series3
Output: Utah       1.0
        Ohio       4.0
        Texas      7.0
        Oregon    10.0
        Name: d, dtype: float64
```

```
frame.sub(series3, axis="index")
Output:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

Function Application and Mapping

- NumPy ufuncs (element-wise array methods) also work with pandas objects

```
frame = pd.DataFrame(np.random.standard_normal((4, 3)),
                     columns=list("bde"),
                     index=["Utah", "Ohio", "Texas", "Oregon"])

frame
Output:
               b         d         e
Utah    1.369949  1.117721 -0.645598
Ohio    1.374224  0.544091  1.353410
Texas  -0.132665 -0.382728  0.333791
Oregon -1.126117  1.727428  0.199735

np.abs(frame)
Output:
               b         d         e
Utah    1.369949  1.117721  0.645598
Ohio    1.374224  0.544091  1.353410
Texas   0.132665  0.382728  0.333791
Oregon  1.126117  1.727428  0.199735
```

Function Application and Mapping

- Another frequent operation is **applying a function** on one-dimensional arrays to each column or row
  - DataFrame's **apply** method does exactly this
  - Here the function **f**, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in **frame**
  - The result is a Series having the columns of **frame** as its index
  - If we pass **axis="columns"** to **apply**, the function will be invoked once per row instead
    - A helpful way to think about this is as **"apply across the columns"**

```
1  def f1(x):
2    return x.max() - x.min()
3
4  frame.apply(f1)
5  Output: b    2.500340
6          d    2.110156
7          e    1.999008
8          dtype: float64
```

```
1  frame.apply(f1, axis="columns")
2  Output: Utah      2.015547
3          Ohio      0.830133
4          Texas     0.716519
5          Oregon    2.853545
6          dtype: float64
```

Function Application and Mapping

- **Note**: Many of the most common array statistics (like **sum** and **mean**) are DataFrame methods, so using **apply** is *not necessary*
- The *function* passed to **apply** need not return a scalar value; it can also return a Series with multiple values

```
def f2(x):
  return pd.Series([x.min(), x.max()], index=["min", "max"])

frame.apply(f2)
Output:
             b         d         e
min -1.126117 -0.382728 -0.645598
max  1.374224  1.727428  1.353410
```

Function Application and Mapping

- Element-wise Python functions can be used, too
  - Suppose we wanted to compute a formatted string from each floating-point value in **frame**
  - We can do this with **applymap**
  - The reason for the name **applymap** is that Series has a **map** method for applying an element-wise function

```
def my_format(x):
  return f"{x:.2f}"

frame.applymap(my_format)
Output:
              b      d      e
Utah       1.37   1.12  -0.65
Ohio       1.37   0.54   1.35
Texas     -0.13  -0.38   0.33
Oregon    -1.13   1.73   0.20
```

```
frame["e"].map(my_format)
Output: Utah      -0.65
        Ohio       1.35
        Texas      0.33
        Oregon     0.20
        Name: e, dtype: object
```

Sorting and Ranking

- Sorting a dataset by some criterion is another important built-in operation
  - To sort lexicographically by row or column label, use the **sort_index** method
  - Returns a new, sorted object

```
obj = pd.Series(np.arange(4),
                index=["d", "a", "b", "c"])

obj
Output: d    0
        a    1
        b    2
        c    3
        dtype: int32
```

```
obj.sort_index()

Output: a    1
        b    2
        c    3
        d    0
        dtype: int32
```

Sorting and Ranking

- With a DataFrame, we can sort by index on either axis

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=["three", "one"],
                     columns=["d", "a", "b", "c"])

frame
Output:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

frame.sort_index()
Output:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

frame.sort_index(axis="columns")
Output:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

Sorting and Ranking

- The data is sorted in **ascending** order by **default** but can be sorted in *descending* order, too

```
frame.sort_index(axis="columns", ascending=False)
Output:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Sorting and Ranking

- To sort a **Series** by its values, use its **sort_values** method

```
obj = pd.Series([4, 7, -3, 2])

obj.sort_values()
Output: 2   -3
        3    2
        0    4
        1    7
        dtype: int64
```

- Any **missing values** are sorted to the **end** of the Series by default

```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

obj.sort_values()
Output: 4   -3.0
        5    2.0
        0    4.0
        2    7.0
        1    NaN
        3    NaN
        dtype: float64
```

Sorting and Ranking

- Missing values can be sorted to the start instead by using the **na_position** option

```
obj.sort_values(na_position="first")

Output: 1    NaN
        3    NaN
        4   -3.0
        5    2.0
        0    4.0
        2    7.0
        dtype: float64
```

Sorting and Ranking

- When sorting a DataFrame, we can use the data in one or more columns as the sort keys
  - To do so, pass one or more column names to **sort_values**

```
frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})

frame
Output:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1

frame.sort_values("b")
Output:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

Sorting and Ranking

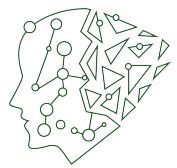
- To sort by *multiple columns*, pass a list of names

```
frame.sort_values(["a", "b"])

Output:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

Sorting and Ranking

- **Ranking** assigns ranks from one through the number of valid data points in an array, starting from the lowest value
  - The **rank** methods for Series and DataFrame are the place to look; by default, **rank** breaks ties by assigning each group the mean rank

```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

obj.rank()
Output: 0    6.5
        1    1.0
        2    6.5
        3    4.5
        4    3.0
        5    2.0
        6    4.5
        dtype: float64
```



Sorting and Ranking

- Ranks can also be assigned according to the order in which they're observed in the data
  - Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data

```
obj.rank(method="first")
Output: 0    6.0
        1    1.0
        2    7.0
        3    4.0
        4    3.0
        5    2.0
        6    5.0
        dtype: float64
```

Sorting and Ranking

- Can rank in descending order, too

```
obj.rank(ascending=False)

Output: 0    1.5
        1    7.0
        2    1.5
        3    3.5
        4    5.0
        5    6.0
        6    3.5
        dtype: float64
```

Sorting and Ranking

- DataFrame can compute ranks over the rows or the columns

```
frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
                      "c": [-2, 5, 8, -2.5]})

frame
Output:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5

frame.rank(axis="columns")
Output:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

Sorting and Ranking

- **Tie-breaking** methods with rank:

Method	Description
`"average"`	Default: assign the average rank to each entry in the equal group
`"min"`	Use the minimum rank for the whole group
`"max"`	Use the maximum rank for the whole group
`"first"`	Assign ranks in the order the values appear in the data
`"dense"`	Like `method="min"`, but ranks always increase by 1 between groups rather than the number of equal elements in a group

Axis Indexes with Duplicate Labels

- While many pandas functions (like **reindex**) require that the labels be **unique**, it's not mandatory
  - Consider a small Series with duplicate indices
  - The **is_unique** property of the index can tell us whether or not its labels are unique

```
obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])

obj
Output: a    0
        a    1
        b    2
        b    3
        c    4
        dtype: int32

obj.index.is_unique
Output: False
```

Axis Indexes with Duplicate Labels

- Data selection is one of the main things that behaves differently with duplicates
  - Indexing a label with **multiple entries** returns a *Series*
  - **Single entries** return a *scalar value*
  - This can make our code more complicated, as the output type from indexing can vary based on whether or not a label is repeated

```
obj["a"]
Output: a    0
        a    1
        dtype: int32

obj["c"]
Output: 4
```

Axis Indexes with Duplicate Labels

- The same logic extends to indexing rows (or columns) in a **DataFrame**

```
df = pd.DataFrame(np.random.standard_normal((5, 3)),
                  index=["a", "a", "b", "b", "c"])

df
Output:
          0         1         2
a  0.325674  0.577541  0.010558
a -0.830407  0.323302 -0.463249
b  0.636487 -0.952166 -0.184723
b  0.338095 -0.283760 -0.665370
c -1.947084  0.112052 -1.016767

df.loc["b"]
Output:
          0         1         2
b  0.636487 -0.952166 -0.184723
b  0.338095 -0.283760 -0.665370

df.loc["c"]
Output: 0   -1.947084
        1    0.112052
        2   -1.016767
        Name: c, dtype: float64
```

Table of Contents

Summarizing and Computing Descriptive Statistics

- pandas objects are equipped with a set of common **mathematical** and **statistical** methods
  - Most of these fall into the category of *reductions* or *summary statistics*
  - These are methods that extract a single value (like the sum or mean) from a Series, or a Series of values from the rows or columns of a DataFrame
- Compared with the similar methods found on NumPy arrays, they have built-in **handling** for **missing data**
  - Calling DataFrame's **sum** method returns a Series containing column sums
  - Passing **axis="columns"** or **axis=1** sums across the *columns* instead

Summarizing and Computing Descriptive Statistics

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                   [np.nan, np.nan], [0.75, -1.3]],
                  index=["a", "b", "c", "d"],
                  columns=["one", "two"])

df
Output:
     one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3

df.sum()
Output: one    9.25
        two   -5.80
        dtype: float64

df.sum(axis="columns")
Output: a    1.40
        b    2.60
        c    0.00
        d   -0.55
        dtype: float64
```

Summarizing and Computing Descriptive Statistics

- When an entire row or column contains all NA values, the sum is 0, whereas if any value is not NA, then the result is NA
  - Can be disabled with the **skipna** option, in which case any NA value in a row or column names the corresponding result NA

```
df.sum(axis="index", skipna=False)

Output: one    NaN
        two    NaN
        dtype: float64
```

```
df.sum(axis="columns", skipna=False)
Output: a     NaN
        b    2.60
        c     NaN
        d   -0.55
        dtype: float64
```

- Some aggregations, like **mean**, require **at least one non-NA** value to yield a value result, so here we have

```
df.mean(axis="columns")

Output: a    1.400
        b    1.300
        c      NaN
        d   -0.275
        dtype: float64
```

Summarizing and Computing Descriptive Statistics

- Options for **reduction methods**:

Method	Description
`axis`	Axis to reduce over; "index" for DataFrame's rows and "columns" for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Summarizing and Computing Descriptive Statistics

- Some methods, like **idxmin** and **idxmax**, return indirect statistics, like the index value where the minimum or maximum values are attained

```
df.idxmax()

Output: one    b
        two    d
        dtype: object
```

- Other methods are **accumulations**

```
df.cumsum()

Output:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

Summarizing and Computing Descriptive Statistics

- Some methods are neither reductions nor accumulations
  - **describe** is one such example, producing multiple summary statistics in one shot
  - **IMPORTANT**: This method is very useful for obtaining the basic statistics regarding the current DataFrame and is used very often

```
df.describe()

Output:
              one       two
count    3.000000  2.000000
mean     3.083333 -2.900000
std      3.493685  2.262742
min      0.750000 -4.500000
25%      1.075000 -3.700000
50%      1.400000 -2.900000
75%      4.250000 -2.100000
max      7.100000 -1.300000
```

Summarizing and Computing Descriptive Statistics

- On **nonnumeric** data, **describe** produces alternative summary statistics

```
obj = pd.Series(["a", "a", "b", "c"] * 4)

obj.describe()
Output: count      16
        unique      3
        top         a
        freq        8
        dtype: object
```

Summarizing and Computing Descriptive Statistics

- **Descriptive** and **summary statistics**:

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value is obtained, respectively; not available on DataFrame objects
`idxmin, idxmax`	Compute index labels at which minimum or maximum value is obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1 (default: 0.5)
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value

Summarizing and Computing Descriptive Statistics

- **Descriptive** and **summary statistics** (cont.):

`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values
`kurt`	Sample kurtosis (fourth moment) of values
`cumsum`	Cumulative sum of values
`cummin`, `cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute first arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

Correlation and Covariance

- Some summary statistics, like **correlation** and **covariance**, are computed from *pairs of arguments*
- The **corr** method of Series computes the **correlation** of the *overlapping*, *non-NA*, *aligned-by-index* values in two Series. Relatedly, **cov** computes the **covariance**
- Using DataFrame's **corrwith** method, you can compute pair-wise correlations between a DataFrame's columns or rows with another Series or DataFrame
  - Passing a DataFrame computes the correlations of matching column names
  - Passing **axis="columns"** does things row-by-row instead
  - In all cases, the data points are aligned by label before the correlation is computed

Unique Values, Value Counts, and Membership

- Another class of related methods extracts information about the values contained in a one-dimensional Series
- Consider this example:

```
obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])

obj
Output: 0    c
        1    a
        2    d
        3    a
        4    a
        5    b
        6    b
        7    c
        8    c
        dtype: object
```

Unique Values, Value Counts, and Membership

- The first function is **unique**, which gives us an array of the unique values in a Series
  - The unique values are NOT necessarily returned in the order in which they first appear, and NOT in sorted order, but they could be sorted after the fact if needed (**uniques.sort()**)

```
uniques = obj.unique()

uniques
Output: array(['c', 'a', 'd', 'b'], dtype=object)
```

Unique Values, Value Counts, and Membership

- Relatedly, **value_counts** computes a Series containing value frequencies
  - The Series is sorted by value in **descending** order by default as a convenience
  - **value_counts** is also available as a top-level pandas method that can be used with NumPy arrays or other Python sequences

```
obj.value_counts()
Output: c    3
        a    3
        b    2
        d    1
        Name: count, dtype: int64

pd.value_counts(obj.to_numpy(), sort=False)
Output: c    3
        a    3
        d    1
        b    2
        Name: count, dtype: int64
```

Unique Values, Value Counts, and Membership

- **isin** performs a vectorized set **membership** check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame

```
obj

Output: 0    c
        1    a
        2    d
        3    a
        4    a
        5    b
        6    b
        7    c
        8    c
        dtype: object
```

```
mask = obj.isin(["b", "c"])

mask
Output: 0     True
        1    False
        2    False
        3    False
        4    False
        5     True
        6     True
        7     True
        8     True
        dtype: bool
```

```
obj[mask]
Output: 0    c
        5    b
        6    b
        7    c
        8    c
        dtype: object
```

Unique Values, Value Counts, and Membership

- Related to **isin** is the **Index.get_indexer** method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values

```
to_match = pd.Series(["c", "a", "b", "b", "c", "a"])
unique_vals = pd.Series(["c", "b", "a"])

indices = pd.Index(unique_vals).get_indexer(to_match)

indices
Output: array([0, 2, 1, 1, 0, 2], dtype=int64)
```

Unique Values, Value Counts, and Membership

- Unique, value counts, and set membership methods:

Method	Description
`isin`	Compute a Boolean array indicating whether each Series or DataFrame value is contained in the passed sequence of values
`get_indexer`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute an array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

Unique Values, Value Counts, and Membership

- In some cases, we may want to compute a *histogram* on multiple related columns in a DataFrame
- Here, the row labels in the result are the distinct values occurring in all of the columns
  - The values are the respective counts of these values in each column

```
data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
                     "Qu2": [2, 3, 1, 2, 3],
                     "Qu3": [1, 5, 2, 4, 4]})

data
Output:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4

data["Qu1"].value_counts().sort_index()
Output: Qu1
        1    1
        3    2
        4    2
        Name: count, dtype: int64
```

```
result = data.apply(pd.value_counts).fillna(0)

result
Output:
    Qu1  Qu2  Qu3
1   1.0  1.0  1.0
2   0.0  2.0  1.0
3   2.0  2.0  0.0
4   2.0  0.0  2.0
5   0.0  0.0  1.0
```

Unique Values, Value Counts, and Membership

- There is also a **DataFrame.value_counts** method, but it computes counts considering each row of the DataFrame as a tuple to determine the number of occurrences of each distinct row
  - In this case, the result has an index representing the distinct rows as a hierarchical index

```
data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})

data
Output:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

data.value_counts()
Output:
a  b
1  0    2
2  0    2
1  1    1
Name: count, dtype: int64
```

The End

THE END!!!